**Applied EPI RF Matching Network Final Report**
**Due Date March 31 2004**
**Dr. W. Alan Doolittle**

The work proposed was to implement an auto-tuning RF matching network for a Applied EPI plasma source. The work originally was to be implemented in a custom microcontroller system based on the Rabbit™ microcontroller family. After program initiation, Applied EPI expressed reservations regarding this approach based on concerns of the lack of experience with this platform and thus, the inability to field and support this implementation. Thus, substantial effort was set forth to re-engineer the tuning network within the proprietary IOC controller format developed by Applied EPI. To this end, this report details both implementations.

**Proprietary IOC Applied EPI Implementation**

In short, the IOC implementation worked to move motors and tune the capacitors in the matching network but fundamental limitations in the IOC environment prevented full implementation. The IOC environment was found to be very unstable with the EPI controller locking up without notice and very frequently. Furthermore, the slow response of the EPI system prevented real time feedback of the motor/plasma conditions and resulted in an in-effective matching system. Furthermore, the Silvermax™ motors requested by EPI to be used were discontinued mid-stream in the project due to patent infringement accusations and thus, it was not possible to get replacement parts necessary to continue this thread of development. It is Dr. Doolittle's opinion that the IOC environment is not appropriate for a matching network application. Never the less, the following was accomplished under this program:

- Physical matching network was designed and shown to match to the Applied EPI plasma source.
- The Silvermax™ motors EPI desired to be used were incorporated on the matching network.
- Front panel code was implemented to move the motors, select tune mode and interface with the user. Example code is supplied in an appendix at the end of this report.
- PID control algorithms were implemented in IOC but the control loop delays inherent in the Applied EPI IOC environment made actual control of plasma's impossible. – this limitation was overcome in the Rabbit™ microcontroller implementation described in the next section.

## Successful Rabbit Microcontroller Implementation

### Executive Summary

The "Inductive Nitrogen Plasma Source" project aims to create an effective impedance matching network by minimizing the net reflected power within the system; this results in a cost-effective method of obtaining molecular nitrogen (N) utilized for semiconductor applications. Initially, the system attempts to inductively light the nitrogen-based plasma; next, an optical sensor monitors the plasma source to determine whether it is lit or not. Once the plasma is lit, the system adjusts the servo motors, which in turn, vary the position of the oil-filled capacitors to the appropriate impedance. Impedance matching can be conducted in one of two ways: manually or automated using a PID controller. Due to time constraints, only manual tuning has been accomplished. Upon achieving successful impedance matching, data logging must be taken into consideration. In order to incorporate data acquisition, a theoretical TCP/IP interface is presented. This project has several economic applications such as:

- Increasing the efficiency of the nitrogen source
- Developing similar systems for other molecular gases (e.g. Oxygen)
- Marketing this product to semiconductor-research groups (BAE, university research)

•

## Table of Contents


| | |
|---|---|
| Introduction | 1 |
| Project Description | 2 |
| Project Goals | 5 |
| Technical Specifications | 6 |
| Design Approach | 6 |
| Design Alternatives & Tradeoffs | 6 |
| Tasks & Schedule | 9 |
| Project Demonstration | 10 |
| Marketing Cost and Analysis | 10 |
| Summary & Conclusions | 15 |
| Bibliography | 17 |
| Impedance Matching System(WEB) | Appendix A |
| Existing Controller Code (WEB) | Appendix B |
| Optical Sensor Description | Appendix C |
| TCP/IP Theoretical Implementation | Appendix D |

## Introduction

This project is designed to create a matching network for a nitrogen plasma source.

Nitrogen typically exists as a molecular compound (N2) and is not reactive in most situations; to achieve a plasma state, the molecules are split into atomic nitrogen. Splitting is typically achieved by applying significant amounts of heat; however, this is not a practical approach for nitrogen so a plasma source is utilized. A plasma is generated by passing an RF current through an inductor, which encompasses the gas; next, once the appropriate magnetic flux is flowing through the gas, the plasma is sparked. It is necessary to get the power to flow through the gas as opposed to reflecting back at the source.

The inductive impedance is determined by the type of gas, its pressure & density, and the natural impedance of the inductor. The impedance matching system consists of: a Smart Star Control Unit, an RF detector, D/A and A/D converters, 8th order Butterworth filter, "Monolithic Photodiode and Single-Supply Transimpedance Amplifier" (optical sensor), 2 servo motors designated to control the impedance of a network of oil-filled capacitors, and 2 power sources. Theoretically, impedance matching is achieved by adjusting the servo motors to synchronize the oil-filled capacitor's impedance to that of the plasma; the impedance of the plasma can be matched manually, or automatically utilizing C code (see Figure 3 and Appendix B). Once the impedance is matched, the plasma can be ignited.

Therefore, to allow manual and automatic impedance matching capacity, this project will implement matching algorithms and a power controller to compensate for the

remaining reflected power. In addition, an optical sensor will be utilized to determine if the plasma is lit. Finally, due to time constraints and unexpected motor communication issues, a data entering & logging process will be implemented theoretically via a TCP/IP interface, see Appendix D.

This device has many advantages over tuning a plasma by hand. It is ability to automatic tune a plasma, its implementation of a TCP/IP interface to monitor the plasma outside of the lab. As well as its ability to assist in manually tuning a plasma by having predefined set points for lighting the plasma, which are programmed into the controller.

## Project Description

To achieve control of the above-mentioned impedance matching system, a digital controller for a nitrogen plasma source is being constructed, Figure 1. Figure 2 shows the schematic of the matching network. In addition, the digital controller, see Appendix A, has been provided; however, its programming was incomplete as were parts of the signal input system for the controller. The plasma source is based on an inductor containing a tube through which molecular nitrogen can flow. The inductor is in parallel with one capacitor and in series with another, forming a matching network. As depicted in Figure 2, a high power RF signal is sent through the circuit. To form an efficient plasma, the RF power source supplies maximum power to the gas & the impedance of the power supply is matched to the plasma source. Matching the impedance is accomplished by using variable capacitors in the matching network, as seen in Appendix A, which are tuned by independent motors. There is also an RF sensor, which provides all the necessary information about the current state of the RF signal, including measuring forward and

reflected power. The RF detector, inductive coil, and capacitors compose the RF network.

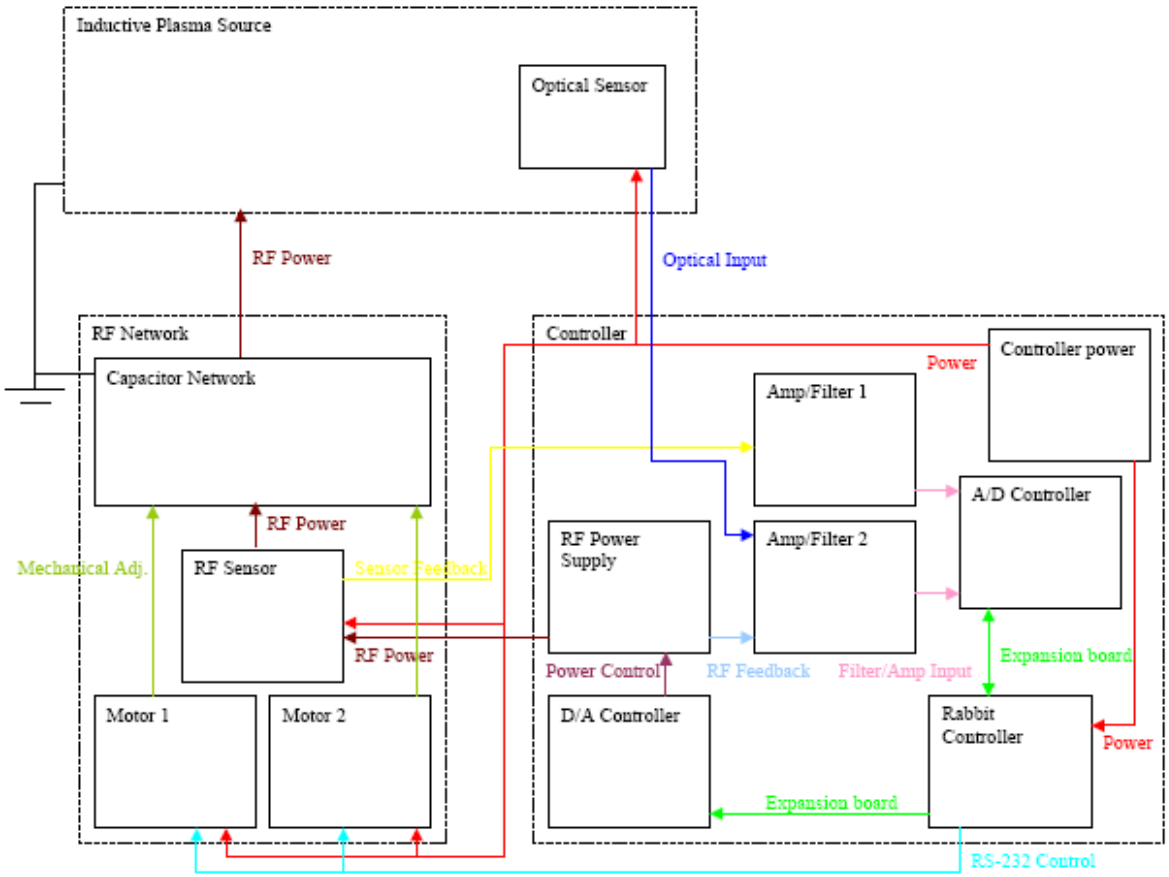
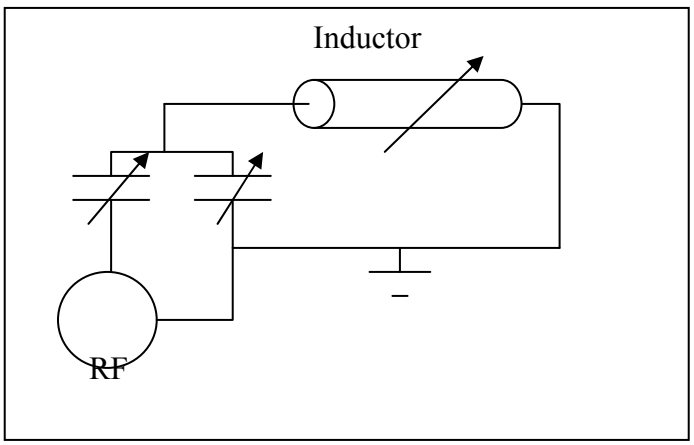



**Figure 1. Layout of matching network with controller.**

All of these devices (digital controller, RF power supply, capacitors, inductor, motors, RF sensor) have been provided prior to the beginning of this project.

There was also a reasonable amount of base programming provided to achieve many of the desired functions which include:

- enabling the controller to communicate with the motors
- acquiring inputs from the controller's A/D converters

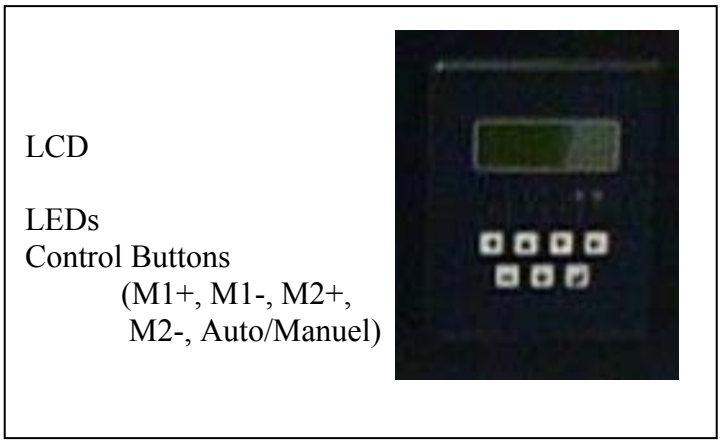


- sending output from D/A converter

The equipment design begins with the filters & amplifiers, which are utilized to condition the inputs from the RF sensor. Next, an optical detector was constructed to indicate whether the plasma is lit, see Appendix C. Also, a clamping diode network was implemented to protect the filter used by the optical detector from any stray power induced by the optical detector's proximity to the plasma source. Once these devices are built, they must be installed in the controller box and wired into the existing system. Once installed, this will conclude the hardware phase of the project.

Upon completion of the hardware phase, software was implemented to control the impedance matching system; there are three phases of software design: manual tuning, automatic tuning, and data acquisition through theoretical TCP/IP interface.

Referring to Figure 3, manual tuning involves slowly ramping the power of the RF power source while manually adjusting the motors (using the LCD buttons) to minimize reflected power. This is similar to most manual controllers in the way it minimizes reflected power. The forward, reflected and total power will be displayed on the LED display so the user can minimize the reflected power. The far left LED will light when a plasma is detected by the optical sensor.




The second phase of the code will allow the controller to automatically tune a plasma based on the inputs from the RF sensor. If the automatic tuning fails, then a push button on the front of the display can be used to switch back to the manual-tuning mode.

Once the user has tuned the plasma, the same button can be pressed to set the controller back to an automatic mode. This implementation can be seen in figure 4.

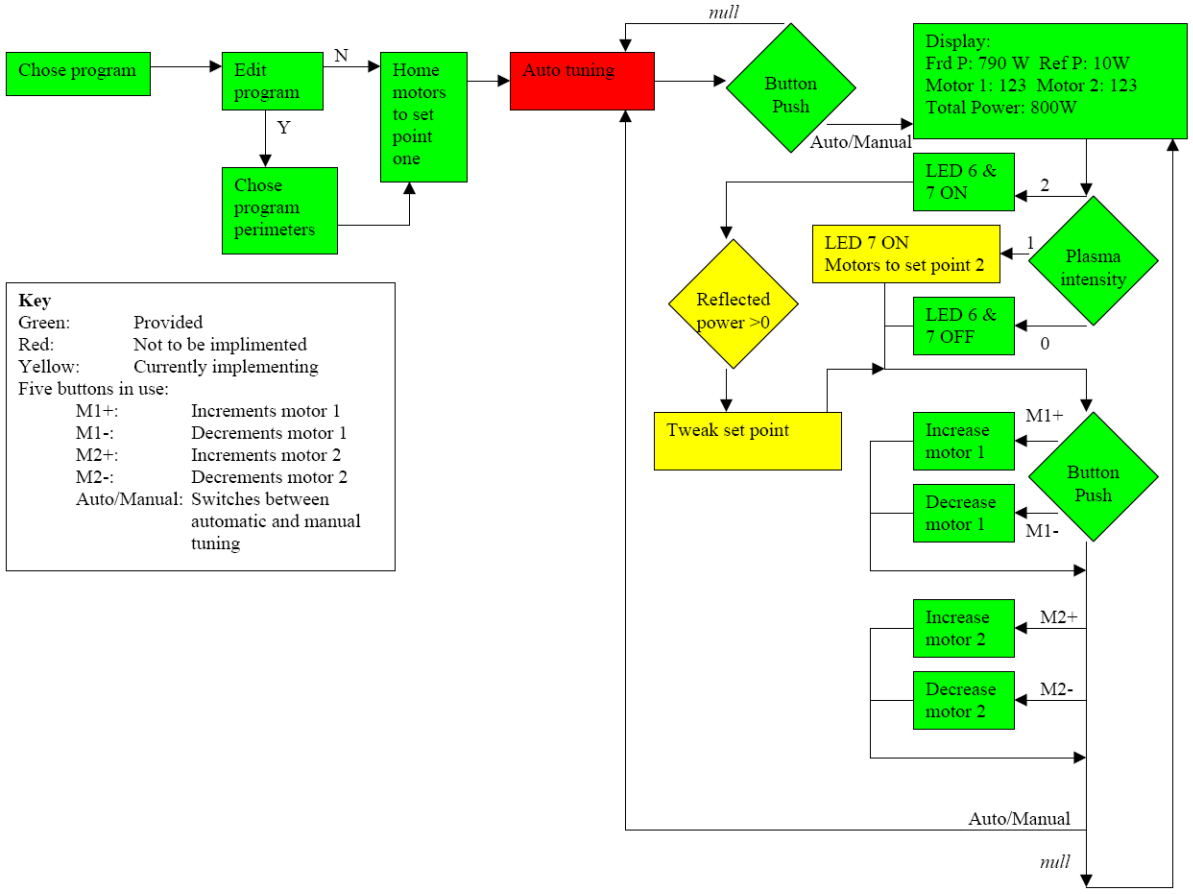



**Fig 4. Software Flowchart.**

The final phase of the software project is the theoretical implementation of a TCP/IP interface on the controller. The controller will be configured to activate the TCP/IP server and a Java applet will be developed to receive, display and log information on the remote computer. In addition, it may be possible to manually tune a plasma remotely; consequently, the user will not have to physically enter the lab to monitor or adjust properties of the impedance matching system.

## Project Goals

The project goals consist of several items, which include, but are not limited to, inductively lighting the plasma, detecting the plasma by utilizing an optical sensor, automated and manual tuning of the impedance matching system, and if time permits, logging the data through a TCP/IP interface.

## Technical Specifications

### Optical Sensor

- SINGLE SUPPLY: +2.7 to +36V
- PHOTODIODE SIZE: 0.090 x 0.090 inch
- HIGH RESPONSIVITY: 0.45A/W (650nm)
- BANDWIDTH: 14kHz at RF = 1M?
- LOW QUIESCENT CURRENT: 120μA

### Filters

- 8TH ORDER BUTTERWORTH MULTIFEEDBACK FILTER
- CUTOFF AT 6Hz
- RESPONSE FREQUENCY 6 Hz
- 1% RESISTORS
- UNITY GAIN
- IN SERIES WITH INSTRUMENTATION AMPLIFIER

### Controller

- 800 watts RF Power supply
- +/-15 volt power supply
- 24 volt power supply
- Rabbit$^{TM}$ Smart Star 9000 controller

**Matching Network**

- Oil filled Capacitors have >80000 settings
- RS232 communication with Servo motors

## Design Approach

The initial design is split into three parts. (1) The design and construction of the filter and amplifier boards. (2) The selection of the optical sensor and the construction of its clamping diode network. (3) The coding of the manual tuning system and later the auto tuning and TCP/IP features. The design of the filter and amplifier boards has been completed using an eighth order Butterworth filter, Figure 5, and the parts are being ordered. An optical detector has also been selected for its compact size and its high selectivity within the infrared range, making plasma detection and signal filtering more effective.

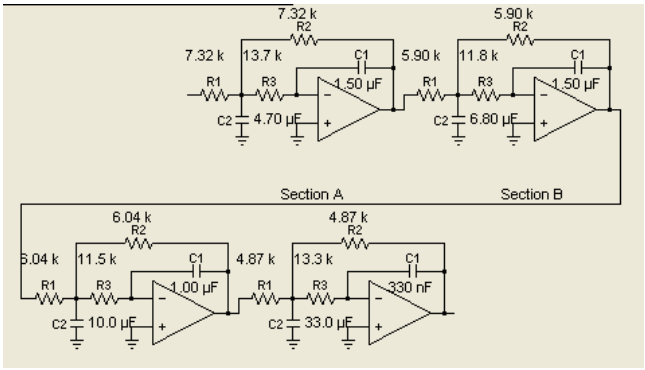



**Figure 5. Eighth order filter.**

The existing design uses two SilverMax servo motors, which are set up in a stepper mode. This configuration was chosen to take advantage of the ease of control of the stepper motor and the speed of the servo motor. All the information about the motors' function and interface are provided in two manuals, "Command Reference" [1] and "User Manual" [2]. The motors are set to communicate in RS-232 mode for easy interface with the Smart Star digital controller. This controller will also receive all the information from the amplifier and filter boards through its A/D converters and will control the RF power supply to ramp power to the system; all specifications can be found

in the Smart Star manual [3]. The Smart Star uses C as its programming language. Finally, the RF sensor, which measures forward and reflected power was obtained from RF Services, Part Number: 232017-04; in addition, the RF sensor provides the majority of the inputs, forward and reflected power and phase and magnitude error, to the controller. Many of the RF sensor's specifications are within its manual [4] except for the range of the output values. Pictures of these components can be seen in Appendix A. The website contains the current C code for the system.

The three preliminary steps: (1) manual control coding, (2) filter design and construction, and (3) optical filter design and implementation, must all be complete before testing of the manual code can commence. Once the boards have been wired into the controller, Figure 1, it will be possible to calibrate the received signals in the code and test the manual code, Figure 3, on the plasma source. Although testing can be conducted without the optical sensor, this would require significantly more coding time. It was planed that once testing and calibration is complete with the manual tuning code, work on the automatic tuning code can begin simultaneously with the theoretical TCP/IP implementation. Due to problems with the system on which the plasma controller operates the controller has not been calibrated so coding of the automatic tuning aspect has been put on hold. The theoretical implementation of TCP/IP interface has continued though.

There have been very few tradeoffs at this time since, as the budget demonstrates, over 99% of the costs of parts are incurred by existing components. Consequently, while deciding which of the two optical detectors (Digikey part #s: opt301, opt101) the opt101 was selected since it was a less expensive, newer device that offered the same

functionality as the opt301. It was also decided to prioritize the implementation of the optical detector in order to minimize the required code for plasma detection.

## Design Alternatives and Tradeoffs

There were a few design tradeoffs in our project that need to be addressed. First was the decision to use active filters rather than passive ones to filter the analog inputs. Then it was decided to use an optical sensor to detect plasma intensity. The final decision was to use a clamping diode network like that used in the telecom industry.

The active filters were used for a number of reasons. First the passive filters would have worked poorly with the output impedance of the detectors, about 75 kilo ohms. The 6 Hz cutoff was chosen so that the DC signal would not be attenuated while removing as much white noise from the plasma as possible. Finally, the boards, which the filters were built on, were provided and instructions were given to use them.

The optical sensor was chosen since it was inexpensive and easy to implement while reducing the complexity of the code. It allowed the system to optically detect a plasma rather than using impedance calculations to do so. This feature also gave the system the ability to give the user an actual plasma intensity, which is useful to know relativity how much atomic oxygen being produced.

## Summary & Conclusions

The design of the inductive nitrogen plasma source, in summary, is composed of three main components: inductive plasma source, controller, and RF network. The RF detector, inductive coil, and capacitors compose the RF network, the op amps, power supplies, rabbit controller, clamping diode network, and filters are in the controller, and

the optical sensor is found within the plasma source module. Also, an important goal, which was successfully completed, was to incorporate the optical sensor and op-amp/filters into the existing hardware.

The filter, an eighth order multiple feedback Butterworth filter, with cutoff frequency at 6 Hz, was designed. In conjunction with designing the filter, an appropriate optical sensor was chosen to fit our application. Furthermore, the designs for the op-amps were completed and reviewed. Next, construction of op-amp boards began and is completed. Simultaneously with the above-mentioned tasks (designing filter, choosing optical sensor, ordering parts) ongoing work was conducted on coding, which consists of debugging the existing motor control systems and modifying function calls as necessary. In addition, due to time constraints, a theoretical TCP/IP implementation was researched, refer to Appendix D.

Based on the application of any component within the overall system, a complex, expensive device is NOT necessarily superior to the simpler, less expensive device. (i.e. "Always strive to utilize the most practical implementation, not necessarily the most sophisticated one" - Moatasm) in regards to the optical sensor. However, more expensive filters were necessary in designing the filters and disused in the design tradeoffs.

The project demonstration went well. All the components operated within desired parameters.

The RF sensor outputs have negative and positive outputs which were designed to range from -15 volts to 0 volts and +15 volts to 0 volts respectively, but require a range from -15 volts to +15 volts.

## Bibliography

[1] "SilverMax Command Reference." QuickSilver Controls, Inc., Tech. Rep. Revision 4.00, Dec, 2002.

[2] "SilverMax User Manual." QuickSilver Controls, Inc., Tech. Rep. Revision 3.21, Mar, 2002.

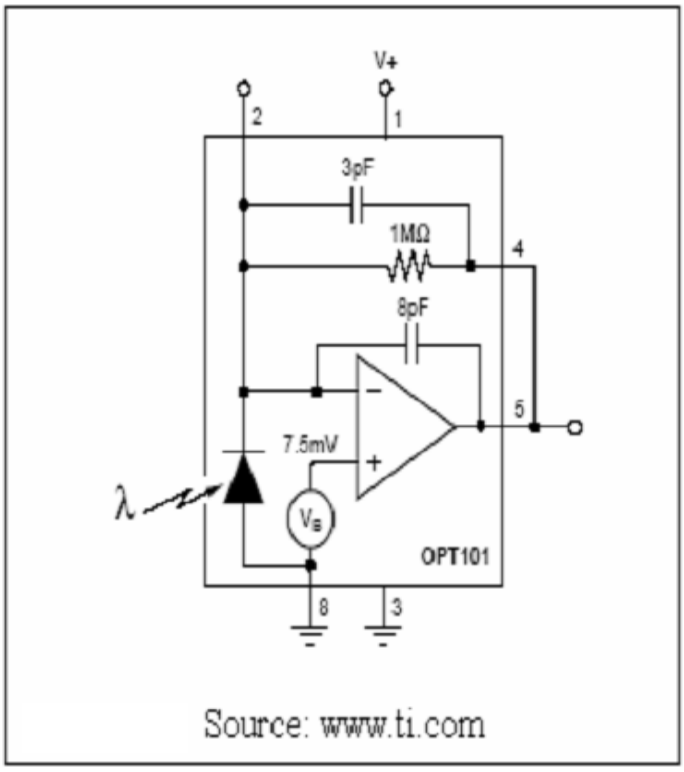
[3] "Smart Star (SR9000) User's Manual." Z-World, Inc., Davis, CA Tech. Rep. Part Number 019-0107, 020301-A, 2002.

[4] "Manual for RF Input Detector RFS Part Number: 232017-04." RF Services, Inc., Sunnyvale, CA Tech. Rep. Document No. 232527-04 Rev A, 1998.

# Appendix A:
# Optical Sensor Description

Purpose:

Optical sensor utilized to detect whether plasma is lit or not.



Schematic:

- **Pin 1: V+ = 15 V**
- **Pins 4 & 5 shorted; (output pins)**
- **Pins 3 & 8 grounded**
- **NOT USED: Pins 2**

**Testing:**

**To test the operation of the optical sensor, a voltmeter was connected to the output & an observance of the sensor's response to different ambient light stimuli (light / dark) revealed:**

**- Pin 5 connected alone produced digital output (0 V or 15 V)**

**- Pin 4 & 5 shorted results in analog output range (0 - 15 V)**

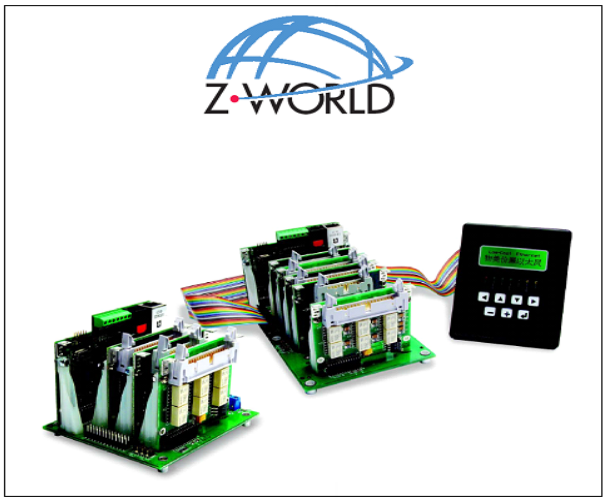
**Conclusion:**

**Analog output allows user to differentiate between a bright inductively-coupled plasma and a dim capacitively-coupled one.**

Appendix B:
TCP/IP Theoretical Implementation

Purpose:

A TCP/IP interface provides remote data input and acquisition capability.

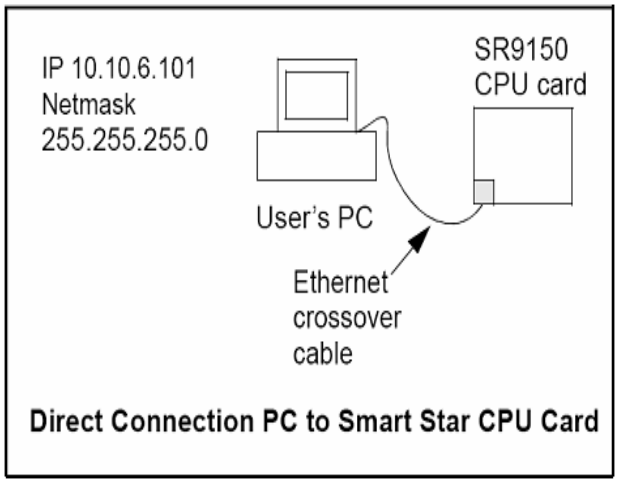



Specifications:

- **Smart Star (SR9000)**
  **- our controller (top picture)**
- **Contains a TCP/IP port**

**(bottom picture)**

- **Network can be setup as VPN ("Virtual Private Network") or via Internet**
- **When a direct connection is made, the interface must perform "ping" test:**



Direct Connection PC to Smart Star CPU Card

**- green LNK light on CPU card: Ethernet connected**
**- orange ACT light: data transfer**

- **"Ping test" and upcoming Smart Star integrated programs (i.e. provided by manufacturer) are run in "Dynamic C"**

**<u>Integrated programs for TCP/IP implementation:</u>**

- **"SSI.C" -** *how to make Smart Star CPU card a web server; allows user to remotely turn the LEDs on & off*
- **"TCP_RESPOND.C" & "TCPSEND.C" -**

*allows the remote user to:*

*a) send data from the remote program to the Smart Star CPU card*

*b) receive data from the CPU card*

**Integrated programs for TCP/IP implementation (cont):**

- **"echo.c" -** *Demonstrates the **tcp_listen()** call. A basic server, that when a client connects, echoes back to the client (user) any data that they send. Can be configured to listen on different TCP ports.*
- **"state.c" -** *Demonstrates building a state machine-based Internet server (i.e. Java Applet). According to zworld.com: prior to using this program, "study the usage of sock_dataready() and sock_fastwrite() here when implementing your own server".*
- **"ftp_client.c" -** *This program utilizes the Rabbit FTP client library. It passes all your parameters to ftp_client_setup() [in the form of a file], which opens a connection, and the library will download the file. If successful it displays the file, otherwise the error code. Therefore, this program can be used to upload a file to the existing impedance matching system thereby allowing the user to remotely input data parameters (i.e. motor positions)*
- **"dac-ctrl.c"-** *Provides a Java applet to control DAC's (digital analog converters). Includes Java source and byte-code files.*
- **Reference:** [3] in Bibliography
- **Tech Support:** www.zworld.com/support

**Direct Connection Procedure Overview**
**(details in [3] Section 5.2):**

- **5.2.1 How to Set IP Addresses in the Sample Programs**

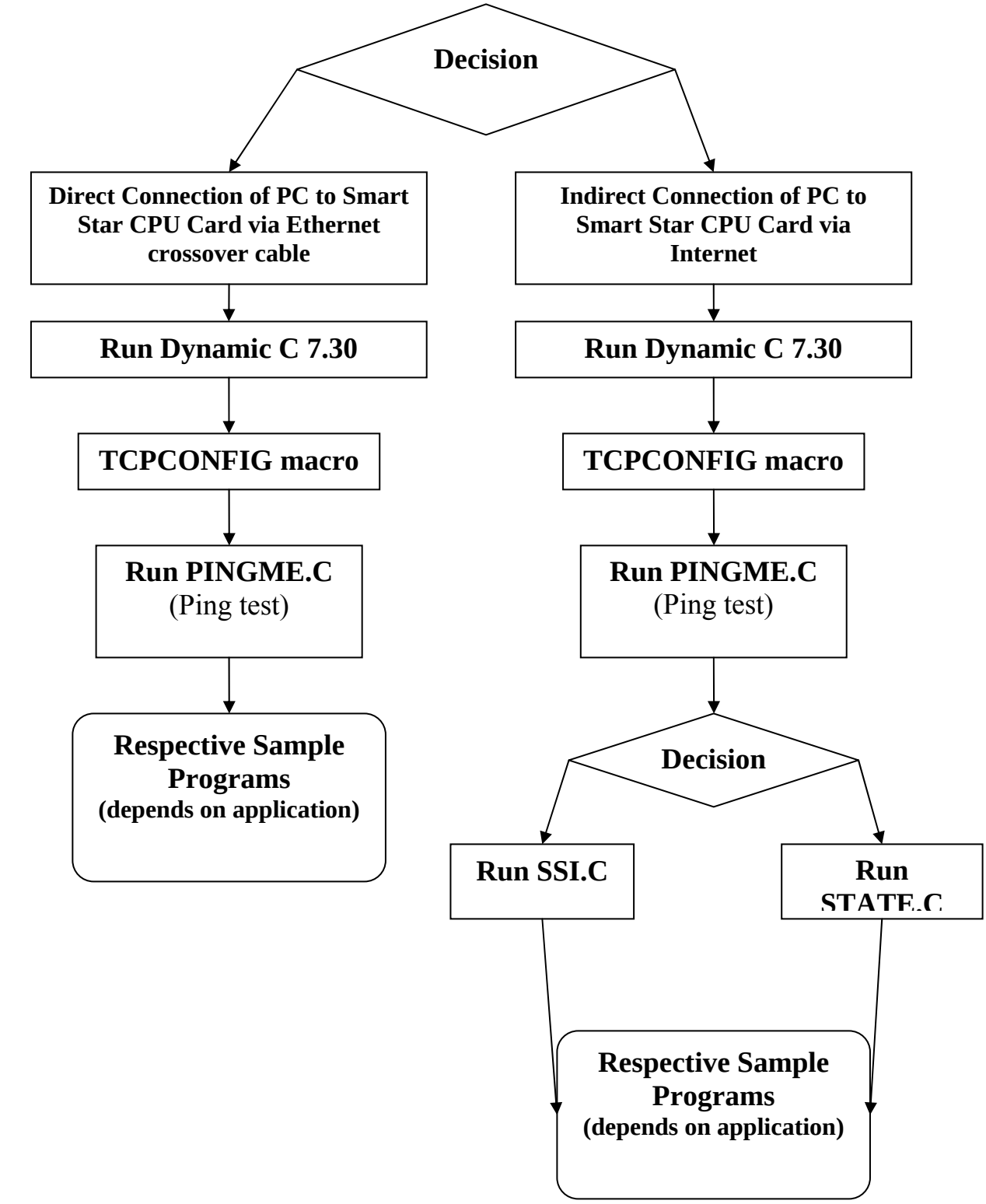
*"With the introduction of Dynamic C 7.30, we [Z-World©] have taken steps to make it easier to run many of our sample programs. You will see a **TCPCONFIG** macro. This macro tells Dynamic C to select your configuration from a list of default configurations." [3]*

- **5.2.2 How to Set Up your Computer's IP Address for a Direct Connection**

**"***Click on Start > Settings > Control Panel to bring up the Control Panel, and then double-click the Network icon. Depending on which version of Windows you are using, look for the TCP/IP Protocol/Network > Dial-Up Connections/Network line or tab. Double-click on this line or select Properties or Local Area Connections > Properties to bring up the TCP/IP properties dialog box. You can edit the IP address and the subnet mask directly. (Disable "obtain an IP address automatically".) You may want to write down the existing values in case you have to restore them later. It is not necessary to edit the gateway address since the gateway is not used with direct connect" [3]*

- **5.2.3 Run the PINGME.C Demo**

**Flow Chart of theoretical TCP/IP implementation (process flow from top to bottom):**


Decision
Direct Connection of PC to Smart Star CPU Card via Ethernet crossover cable
Indirect Connection of PC to Smart Star CPU Card via Internet
Run Dynamic C 7.30
Run Dynamic C 7.30
TCPCONFIG macro
TCPCONFIG macro
Run PINGME.C (Ping test)
Run PINGME.C (Ping test)
Respective Sample Programs (depends on application)
Decision
Run SSI.C
Run STATE.C
Respective Sample Programs (depends on application)

# Appendix C – Front Panel Control Software

```
//  Georgia Tech keypad control sequence based on code developed by Ralf Hermann
//  This sequence uses the same branch/return control as the main motion sequence
//  Be sure to read precautions in motion control sequence before modifying

declare (KeypadControlLoopIndex, 0)
//Start with system in manual mode (0) for development.  Change to auto (1) for runtime
version
declare (vManual_Auto, 0)
declare (vLCD, help)
declare (vPositiveManualStepSize, 500)
declare (vNegativeManualStepSize, -500)
declare (vCurrentMotor,1)
declare (vAutoPreset1)
declare (vAutoPreset2)
declare (vPreset11)
declare (vPreset12)
declare (vPreset21)
declare (vPreset22)
declare (vPreset31)
declare (vPreset32)

state(stateKeys)
   ifGet(FrontPanelKey1, YES, stateLED1Preset1Change, "")
   ifGet(FrontPanelKey2, YES, stateLED2Preset2Change, "")
   ifGet(FrontPanelKey3, YES, stateLED3Preset3Change, "")
   ifGet(FrontPanelKey4, YES, stateLED4SaveCurrentStateChange, "")
   ifGet(FrontPanelKey5, YES, stateLED5IgniteModeChange, "")
   ifGet(FrontPanelKey6, YES, stateLED6ExtinguishModeChange, "")
   ifGet(FrontPanelKey7, YES, stateLED7ReduceManualStepSizeChange, "")
   ifGet(FrontPanelKey8, YES, stateLED8IncreaseManualStepSizeChange, "")
   ifGet(FrontPanelKeyPlus, YES, stateAddPosition, "")
   ifGet(FrontPanelKeyMinus, YES, stateSubPosition, "")
// NOTE:  The key labeled FrontPanelAxisSelect actually controls the Manual to
Autotune function
   ifGet(FrontPanelAxisSelect, YES, stateManualAutoToggle, "")
   ifGet(FrontPanelAxisNext, YES, stateToggleCurrentMotor, "")
// Sub-states below stateKeys that service the keypad branch back to stateKeys
// Thus, all keys are poled and serviced before returning to MainControlLoop
//  go(MainControlLoop)
end

//
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<<<<<
<<<<<<<<<<<<<<<<
```

```
//
>>>
```

```
//
>>>
```

```
//
>>>
```

```
   setIo(FrontPanelLED2, ON)
   go(stateKey2Wait)
end

state(stateLED2Off)
   setIo(FrontPanelLED2, OFF)
   go(stateKey2Wait)
end

state(stateKey2Wait)
   ifGet(FrontPanelKey2, NO, stateKeys, "")
end


// **** LED 3 ****

state(stateLED3Change)
   setIo(FrontPanel_LCD, "KEY 3 PRESS")
   ifGet(FrontPanelLED3, OFF, stateLED3On, "")
   go(stateLED3Off)
end

state(stateLED3On)
   setIo(FrontPanelLED3, ON)
   go(stateKey3Wait)
end

state(stateLED3Off)
   setIo(FrontPanelLED3, OFF)
   go(stateKey3Wait)
end

state(stateKey3Wait)
   ifGet(FrontPanelKey3, NO, stateKeys, "")
end


// **** LED 4 ****

state(stateLED4Change)
   setIo(FrontPanel_LCD, "KEY 4 PRESS")
   ifGet(FrontPanelLED4, OFF, stateLED4On, "")
   go(stateLED4Off)
end

state(stateLED4On)
```

```
   setIo(FrontPanelLED4, ON)
   setIo(FrontPanelLED5, OFF)
   setIo(FrontPanelLED6, OFF)
   setIo(FrontPanelLED7, OFF)
   setIo(FrontPanelLED8, OFF)
   // Toggle State on setIo(MT1AS_Velocity, 3000)
   go(stateKey4Wait)
end

state(stateLED4Off)
   setIo(FrontPanelLED4, OFF)
   // Toggle State off setIo(MT1AS_Velocity, 0)
   go(stateKey4Wait)
end

state(stateKey4Wait)
   ifGet(FrontPanelKey4, NO, stateKeys, "")
end


// **** LED 5 ****

state(stateLED5Change)
   setIo(FrontPanel_LCD, "KEY 5 PRESS")
   ifGet(FrontPanelLED5, OFF, stateLED5On, "")
   go(stateLED5Off)
end

state(stateLED5On)
   setIo(FrontPanelLED4, OFF)
   setIo(FrontPanelLED5, ON)
   setIo(FrontPanelLED6, OFF)
   setIo(FrontPanelLED7, OFF)
   setIo(FrontPanelLED8, OFF)
   // Toggle State on setIo(MT1AS_Velocity, 1500)
   go(stateKey5Wait)
end

state(stateLED5Off)
   setIo(FrontPanelLED5, OFF)
   // Toggle State off setIo(MT1AS_Velocity, 0)
   go(stateKey5Wait)
end

state(stateKey5Wait)
   ifGet(FrontPanelKey5, NO, stateKeys, "")
```

```
end


// **** LED 6 ****

state(stateLED6Change)
   setIo(FrontPanel_LCD, "KEY 6 PRESS")
   ifGet(FrontPanelLED6, OFF, stateLED6On, "")
   go(stateLED6Off)
end

state(stateLED6On)
   setIo(FrontPanelLED4, OFF)
   setIo(FrontPanelLED5, OFF)
   setIo(FrontPanelLED6, ON)
   setIo(FrontPanelLED7, OFF)
   setIo(FrontPanelLED8, OFF)
   // Toggle State on setIo(MT1AS_Velocity, 500)
   go(stateKey6Wait)
end

state(stateLED6Off)
   setIo(FrontPanelLED6, OFF)
   // Toggle State off setIo(MT1AS_Velocity, 0)
   go(stateKey6Wait)
end

state(stateKey6Wait)
   ifGet(FrontPanelKey6, NO, stateKeys, "")
end


// **** LED/Button 7 Reduce the Manual Step Size by a factor of 10 ****
   state(stateLED7ReduceManualStepSizeChange)
   modVar(vPositiveManualStepSize,*,0.1)
   modVar(vNegativeManualStepSize,*,0.1)
   // Blink LED to acknowledge action completed
   setIo(FrontPanelLED7, ON)
   setIo(FrontPanelLED7, OFF)
   go(stateKey7Wait)
end
state(stateKey7Wait)
   ifGet(FrontPanelKey7, NO, stateKeys, "")
end
```

```
// **** LED/Button 8 Increase the Manual Step Size by a factor of 10 ****
   state(stateLED8IncreaseManualStepSizeChange)
   modVar(vPositiveManualStepSize,*,10)
   modVar(vNegativeManualStepSize,*,10)
   // Blink LED to acknowledge action completed
   setIo(FrontPanelLED8, ON)
   setIo(FrontPanelLED8, OFF)
   go(stateKey8Wait)
end
state(stateKey8Wait)
   ifGet(FrontPanelKey8, NO, stateKeys, "")
end
```

## Appendix D – PID Motion Control Code

```
// Georgia Tech PID Motion Sequence
//
// Created by Alan Doolittle
//

//State Sequence Control Parameters
  declare (ControlLoopIndex, 0)
  declare (DisplayLoopCounter, 0)
  //Start with system in manual mode (0) for development.  Change to auto (1) for runtime
version
  declare (Manual_Auto, 0)

//Motor Parameters
   //Target Index for motor 1
     declare (vTarget_Position1, 44000)
   //Actual Index for motor 1
     declare (vCurrentPosition1, 0)
   //Step value for motor 1
     declare (vStep1, 12000)
   //Target Index for motor 2
     declare (vTarget_Position2, 44000)
   //Actual Index for motor 2
     declare (vCurrentPosition2, 0)
   //Step value for motor 1
     declare (vStep2, 12000)

// PID Parameters:  Prop unit is %, Int unit is seconds, Der unit is fraction of int time
//                NOTE: Der should never exceed 0.25 or the system will begin to oscillate
   // PID for Motor 1:
     declare (Prop1, 5.0)
     declare (Int1, 5.0)
     declare (Der1, 0.125)
     declare (OutputPercentage1, 0.0)
   // PID for Motor 2:
     declare (Prop2, 5.0)
     declare (Int2, 5.0)
     declare (Der2, 0.125)
     declare (OutputPercentage2, 0.0)

// Plasma Variables
  // Track Plasma Condition: 0=plasma off, 1=plasma on, -1=plasma extinguish detected,
2=plasma ignition detected
  declare (vIgnitionCondition, 0)
  declare (vForwardPower,100.0)
```

```
  declare (vReversePower,1.0)
  declare (vPhaseError,5.0)
  declare (vMagnitudeError,2.0)
  declare (vReflectionCoef,100.0)
  declare (vVSWR,100.0)

//Display Variables
  declare (vLCD_String,"EPI RF Matching Network Ver. 1.1")
  declare (vLCD_BlankString,"                    ")

//
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<<<<<<<
<<<<<<<<<<<<<<<<<<<<<<<<
// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> stateMainLoop
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
//
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>><<<<<<<<<<<<<<<<<<<<<<
<<<<<<<<<<<<<<<<<<<<<<<<
state(stateMainLoop)
// This sequence handles the main switching/looping between states.
// The loops are repeated by branching to various states.  The state
// then increments the ControlLoopIndex and branches back to mainloop.
// Using this approach, be sure that all branchs and sub-branches away
// from this MainLoop end in an increment to ControlLoopIndex and then return to
MainLoop

  ifVar(ControlLoopIndex,<,0, stateReadConfigInformation, "Reading and updating
Configuration data")
  ifVar(ControlLoopIndex,<,1, stateStartupSequence, "Initializing and Homing motors")
  ifVar(ControlLoopIndex,<,2, stateDetermineKeyStatus, "Checking Keypad")
  ifVar(ControlLoopIndex,<,3, stateReadVoltages, "Reading RF Sensor Voltages")
  ifVar(ControlLoopIndex,<,4, stateCalculatePID_Values, "Updating PID parameters")
  ifVar(ControlLoopIndex,<,5, stateUpdateMotorPositions, "Updating Motor Positions")
  ifVar(ControlLoopIndex,<,6, stateOutputControlVoltages, "Outputing RF Power
Control Voltages")

// Since updating the display is low priority
// only update the display every 5 iterations
  modVar(DisplayLoopCounter,+,1)
  ifVar(DisplayLoopCounter,>4,stateUpdateDisplay,"")

// Loop through again skipping initialization steps
  modVar(ControlLoopIndex,=,2)
end
```

```
//
>>>
```

```
    getIo(MT2GS_Position, vCurrentPosition2)
    getIo(MT1GS_Position, vCurrentPosition1)
    // If the proper zeroposition was previously achieved,
    // current positions should be close to 0

    //Try to move the motors 22 turns in very slow mode
    setIo(MT1AS_RelativePositionSlow, -88000)
    setIo(MT2AS_RelativePositionSlow, -88000)
    // Wait enough time for the motors to reach their limits
    // Note this delay must be set to ~24 if cap position is random.
    // During development, this can be lowered to speed up initialization time
    delay(14)

// Should set index to zero for each loop.  However, this is not currently working.
// For 20 turn capacitors, the max index should be 4000 x 20=80,000
    getIo(MT2GS_Position, vCurrentPosition2)
    getIo(MT1GS_Position, vCurrentPosition1
    // If the proper zeroposition was previously achieved,
    // current positions should be close to 0

// ****** Note these routines are not currently working so comment them out *******
// setIo(MT2AS_ZeroMotorController, " ")
// setIo(MT1AS_ZeroMotorController, " ")
//      delay(1)

//  Update the display
    setIo(FrontPanel_LCD, vLCD_BlankString)
    setIo(FrontPanel_LCD, "EPI RF Matching Network Ver. 1.1")

// Initialize the Keypad front panel
    setIo(FrontPanelLED1, OFF)
    setIo(FrontPanelLED2, OFF)
    setIo(FrontPanelLED3, OFF)
    setIo(FrontPanelLED4, OFF)
    setIo(FrontPanelLED5, OFF)
    setIo(FrontPanelLED6, OFF)
    setIo(FrontPanelLED7, OFF)
    setIo(FrontPanelLED8, OFF)

// Increment the mainloop index and return to stateMainLoop
    modVar(ControlLoopIndex,+,1)
    go(stateMainLoop)

end
```

```
// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>> stateReadConfigInformation
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
state(stateReadConfigInformation)
// Read configuration variables from disk
// Future expansion
// Increment the mainloop index and return to stateMainLoop
   modVar(ControlLoopIndex,+,1)
   go(stateMainLoop)
end

// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>> stateDetermineKeyStatus
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
state(stateDetermineKeyStatus)
// Read keypad status
// add Key dispursement routines later
// Increment the mainloop index and return to stateMainLoop
   modVar(ControlLoopIndex,+,1)
   go(stateMainLoop)
end

// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> stateReadVoltages
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
state(stateReadVoltages)
// Read RF Sensor Voltages
// Add later
// Increment the mainloop index and return to stateMainLoop
   modVar(ControlLoopIndex,+,1)
   go(stateMainLoop)
end


// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> stateCalculatePID_Values
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
state(stateCalculatePID_Values)
// Update PID loop parameters
// Add later
// Increment the mainloop index and return to stateMainLoop
   modVar(ControlLoopIndex,+,1)
   go(stateMainLoop)
end

// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> stateUpdateMotorPositions
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
state(stateUpdateMotorPositions)
// Update Motor/Capacitor Positions
   setIo(MT1AS_Position, vTarget_Position1)
```

```
   setIo(MT2AS_Position, vTarget_Position2)
//    delay(0.1)
//    In the final runtime version, the PID algorythm will continuously update
//    and zoom in on the setpoint.  Thus, no wait for position statements are required.
//    In the debug, use a hard delay
      delay(1)
       getIo(MT1GS_Position, vCurrentPosition1)
//       ifVar(vCurrentPosition1, ~=[5], vTarget_Position1, stateWaitForPosition2, "")
       getIo(MT2GS_Position, vCurrentPosition2)
//       ifVar(vCurrentPosition2, ~=[5], vTarget_Position2, stateNextPosition, "")


// Increment the mainloop index and return to stateMainLoop
   modVar(ControlLoopIndex,+,1)
   go(stateMainLoop)
end

// >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> stateOutputControlVoltages
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
state(stateOutputControlVoltages)
// Update RF Power Control Voltages
// Add later
// Increment the mainloop index and return to stateMainLoop
   modVar(ControlLoopIndex,+,1)
   go(stateMainLoop)
end


// This ruotine is not currently being used
state(stateNextPosition)
//   delay(0.2)
  getIo(MT1GS_Position, vCurrentPosition1)
  modVar(vTarget_Position1, +, vStep1)
  modVar(vStep1, *, 0.5)
  getIo(MT2GS_Position, vCurrentPosition2)
  modVar(vTarget_Position2, +, vStep2)
  modVar(vStep2, *, 0.5)
  go(stateUpdateMotorPositions)
end
```